Random vs. Combinatorial Methods for Discrete Event Simulation of a Grid Computer Network

D. Richard Kuhn*, Raghu Kacker*, Yu Lei**
**National Institute of Standards and Technology, ** University of Texas, Arlington*
kuhn@nist.gov, raghu.kacker@nist.gov, ylei@uta.edu

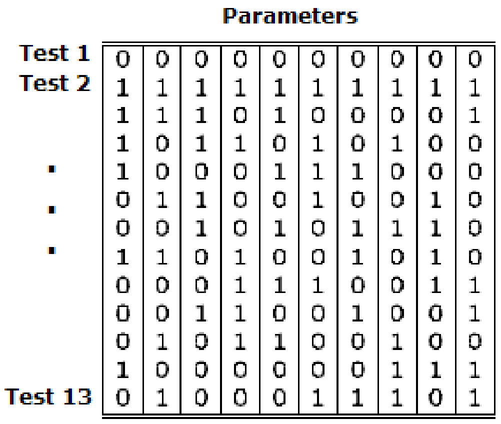
Abstract: This study compared random and *t*-way combinatorial inputs of a network simulator, to determine if these two approaches produce significantly different deadlock detection for varying network configurations. Modeling deadlock detection is important for analyzing configuration changes that could inadvertently degrade network operations, or to determine modifications that could be made by attackers to deliberately induce deadlock. Discrete event simulation of a network may be conducted using random generation of inputs. In this study, we compare random with combinatorial generation of inputs. Combinatorial (or *t*-way) testing requires every combination of any *t* parameter values to be covered by at least one test. Combinatorial methods can be highly effective because empirical data suggest that nearly all failures involve the interaction of a small number of parameters (1 to 6). Thus, for example, if all deadlocks involve at most 5-way interactions between *n* parameters, then exhaustive testing of all *n*-way interactions adds no additional information that would not be obtained by testing all 5-way interactions. While the maximum degree of interaction between parameters involved in the deadlocks clearly cannot be known in advance, covering all *t*-way interactions may be more efficient than using random generation of inputs. In this study we tested this hypothesis for *t* = 2, 3, and 4 for deadlock detection in a network simulation. Achieving the same degree of coverage provided by 4-way tests would have required approximately 3.2 times as many random tests; thus combinatorial methods were more efficient for detecting deadlocks involving a higher degree of interactions. The paper reviews explanations for these results and implications for modeling and simulation.

1 Background

A number of studies have shown combinatorial methods to be highly effective for software testing (e.g., [3],[6],[16],[8]. The effectiveness of combinatorial test methods rests on the observation that a significant number of events in software are triggered only by the interaction of two or more variable values. By including tests for all 2-way, 3-way, etc., interactions, the test set should be able to detect events that occur only with complex interactions. The complexity of discrete event simulation suggests that, as with software testing, combinatorial methods may be effective for finding events triggered only by rare multi-way interactions of input values. In this paper, we compare the effectiveness of combinatorial versus random generation of inputs in a grid computer network simulation for finding configurations that lead to deadlock.

The key enabler in combinatorial testing is a *covering array* that covers all *t*-way combinations of parameter values, for a desired strength *t*. Covering arrays are combinatorial objects that represent interaction test suites. A covering array, $CA(N;t,k,v)$, is an *N* x *k* array, where *k* is the number of variables, and *v* is the number of possible values for each variable such that in every *N* x *t* subarray, each *t*-tuple occurs at least once, then *t* is the *strength* of the coverage of interactions. Each row of a covering array represents a test, with one column for each parameter that is varied in testing. Collectively, the rows of the array include every *t*-way combination of parameter values at least once. For example, Figure 1 shows a covering array that includes all 3-way combinations of binary values for 10 parameters. Each row corresponds to one test, and each column gives the values for a particular parameter. It can be seen that any three columns in any order contain all eight possible combinations (000, 001, 010, 011, 100, 101, 110, 111) of the parameter values. Collectively, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage.

The primary goal in the simulation is to study the behavior of the system with different input configurations. For example, a network simulation may investigate the effect of configurations on packet rate, delay, or potential for deadlock in the network, just as.a production line simulation may study the effects of changing line speed, interconnection between workstations, and buffer size on the number of items that can be produced per hour.



	Parameters									
Test 1	0	0	0	0	0	0	0	0	0	0
Test 2	1	1	1	1	1	1	1	1	1	1
	1	1	1	0	1	0	0	0	0	1
	1	0	1	1	0	1	0	1	0	0
.	1	0	0	0	1	1	1	0	0	0
.	0	1	1	0	0	1	0	0	1	0
.	0	0	1	0	1	0	1	1	1	0
	1	1	0	1	0	0	1	0	1	0
	0	0	0	1	1	1	0	0	1	1
	0	0	1	1	0	0	1	0	0	1
	0	1	0	1	1	0	0	1	0	0
	1	0	0	0	0	0	0	1	1	1
Test 13	0	1	0	0	0	1	1	1	0	1

Figure 1: 3-way covering array for 10 parameters with 2 values each

In this study we compare random and combinatorial testing of a network simulator, to determine if these two test approaches produce significantly different deadlock detection in the simulation. Using deadlocks as events of interest makes evaluating program responses straightforward and unambiguous. Numerical results such as packet rates or delays are not considered, but could be the subject of a future investigation. The two test modes – random or combinatorial – are compared using a standard two-tailed t-test for statistical significance.

2 Experimental Evaluation

This work investigates the hypothesis that combinatorial test suites will detect significantly more deadlocks than random test suites of the same size, for interaction strengths of t = 2, 3, 4.

Independent and Dependent Variables: The independent variable in this study is the type of testing used, either *t*-way combinatorial or random. The dependent variable is the number of deadlocks detected.

Subject Application and Test Suites: Software under test for the experiment was Simured [13], a multicomputer network simulator developed at the University of Valencia. The software is available in C++ and Java versions, for both Linux and Windows. The core command line code (not including user interface or graphical display) consists of 2,131 lines of C++. Simured provides a simulation of the switching and routing layers for a multicomputer, allowing the user to study grid computer configurations to investigate the effect of topologies and configurable parameters on routing, timing, and other variables of interest. We used the C++ command line version of this software, compiled with gcc and run on 64-bit processors under Red Hat Enterprise Linux V4. No modifications were made to the Simured software.

Table 1: Simured configuration parameters and test values used

Parameter		Values
1	DIMENSIONS	1,2,4,6,8
2	NODOSDIM	2,4,6
3	NUMVIRT	1,2,3,8
4	NUMVIRTINJ	1,2,3,8
5	NUMVIRTEJE	1,2,3,8
6	LONBUFFER	1,2,4,6
7	NUMDIR	1,2
8	FORWARDING	0,1
9	PHYSICAL	t, f
10	ROUTING	0,1,2,3
11	DELFIFO	1,2,4,6
12	DELCROSS	1,2,4,6
13	DELCHANNEL	1,2,4,6
14	DELSWITCH	1,2,4,6

Simured provides a set of 14 parameters that can be set to a variety of values in a configuration file that is read by the simulator. Parameters and possible values used are shown in Table 1. The total number of possible configurations with these parameter values is 3.1 x 10^7. Larger values are possible for a number of parameters, but would require extensive run time on a large system.

Evaluation Metrics: Test suites were evaluated according to the number of deadlocks detected. We also compare the percentage of *t*-way combinations covered for the random test suites of equal size, and determine the number of random tests needed to provide 100% coverage of the respective *t*-way combinations. (By definition, a covering array provides 100% coverage of *t*-way combinations.)

Threats to Validity: Clearly there are limitation on the extent to which these results can be generalized to other applications. While previous comparisons of combinatorial and random testing focused on fault detection, this study evaluates these methods with respect to deadlock detection in a simulation. Some implications of this difference are discussed in the analysis of results, in Section 4.2. A second difference is the nature of the software under test. Simured is a small but complex program that is not assumed to have characteristics similar to other application domains. Network simulation requires extensive calculations for statistics such as packet transmission rates and delays, and is not directly comparable to other types of software.

While the issues raised above should be considered in evaluating results, we believe that the experiment has identified a number of factors that can be usefully considered when deciding whether to use random or combinatorial testing for a particular problem.

3 Testing Procedure

Covering arrays that include all *t*-way combinations for *t* = 2, 3, and 4 were generated using the IPOG algorithm [11], which produces compact test suites. Test suites for the configuration shown in 0 included 28, 161, and 752 tests for *t* = 2, 3, and 4 respectively. Random test suites matching the sizes of the 2, 3, and 4-way combinatorial test suites were produced using the standard C library *rand()* function, producing one test at a time with a call to *rand()* for each variable value. In generating random test sets, the *rand()* function was initialized with a call to *srand()* to seed the pseudo-random number generator from the system clock. From these tests, configuration files were generated for Simured and the command line version of Simured invoked with each configuration file.

Each test set was executed for 500, 1000, 2000, 4000, and 8000-packet simulation runs. For combinatorial testing, one test suite run was conducted for each of the five packet counts and three interaction levels (28, 161, and 752 tests, for a total of 4,705 simulations). Random generation produces a different test set with each test generation run. For random testing, eight runs at each combination of packet count and interaction level were conducted (37,640 simulations), and the average deadlock detection calculated.

4 Results and Analysis

4.1 Test Results

Results for the two test modes were compared with a standard t-test for paired samples. Table 2 shows the number of deadlocks detected using tests produced from IPOG versus the average number of deadlocks detected with an equal number of randomly generated tests. Values for random test detection represent the average of eight runs with randomly generated tests at each combination of interaction level and packet count. Table 3 gives the two-tailed probability of a difference between the numbers of deadlocks detected by combinatorial and random testing.

Table 2: Deadlocks, combinatorial vs. random

Deadlocks Detected – combinatorial						
t	Tests	Packets				
		500	1000	2000	4000	8000
2	28	0	0	0	0	0
3	161	2	3	2	3	3
4	752	14	14	14	14	14
Average Deadlocks Detected – random						
t	Tests	Packets				
		500	1000	2000	4000	8000
2	28	0.63	0.25	0.75	0. 50	0. 75
3	161	3.00	3.00	3.00	3.00	3.00
4	752	10.13	11.75	10.38	13.00	13.25

Table 3: t-test results for difference between random and IPOG generated tests

Interaction strength	**Two-tailed probability**
2	.0035
3	.1778
4	.0235

For pairwise testing (*t* = 2), combinatorial testing detected slightly fewer deadlocks than an equal number of random tests, and the difference is statistically significant. At interaction strength *t* = 3 the difference between the two test methods is not statistically significant. At *t* = 4, however, the covering arrays produced by IPOG detected significantly more deadlocks than an equal number of random tests. In the next section we consider some possible reasons for the variation in effectiveness of these two test methods. Two important considerations should be noted about the difference in deadlocks detected:

combinatorial methods found more deadlock configurations, but also consistently found 14 deadlocks for the most complex (4-way) interactions, while there was a great degree of variation among the random configurations.

4.2 Analysis of Results

In considering explanations for the results, we first note that there can be a number of differences between the simulations conducted in this work and software testing in other application domains. In many applications, such as databases or web applications, different parameter values may result in different execution paths within an application, but the amount and complexity of processing is often similar for many different inputs. Network simulation, by contrast, may exhibit wide variations in processing depending on whether the input configuration is a small network of simple topology, or a large, complex one. This difference was observed in widely varying run times (not reported in this paper), and may also contribute to the distribution of deadlocks detected at the three interaction levels. Previous work (see Section 1) has found that increasing values of *t* detect progressively fewer faults, even in cases where combinatorial testing performed no better than random tests. Pairwise testing (*t*=2) often detected 70% to more than 90% of faults, while 3-way tests found roughly 10% to 20% of faults, and 4-way to 6-way tests typically detected less than 5%. This distribution is essentially reversed for the Simured testing (see Table 2), with 0%, 18%, and 82% of deadlocks detected at *t*=2, 3, and 4 respectively. This result is not unexpected. Faults can be triggered by combinations of any of the variables in a program. Even though a large set of variables may be directly or indirectly involved in triggering deadlocks, the set can be expected to be much smaller than the total number of variables in a program. With deadlocks occurring in roughly 2% of simulation runs, larger test sets would be expected to locate more deadlocks.

In addition to the “reverse” relationship between deadlock detection and interaction strength, another interesting finding was that pairwise tests detected slightly fewer deadlocks than the same number of random tests. Careful analysis shows that there is in fact a combinatorial explanation for this result, which we discuss in the remainder of this section.

Because a significant percentage of events can only be triggered by the interaction of two or more variables, one consideration in comparing random and combinatorial testing is the degree to which random testing covers particular *t*-way combinations. Any test set will also cover a certain proportion of possible (*t*+1)-way, (*t*+2)-way, etc. combinations as well. Tables 4 and 5 compare this coverage for the Simured test inputs.

We also analyzed the average percentage of *t*-way combinations covered by 100 randomly generated test sets of the same size as a *t*-way covering array generated by IPOG, for various combinations of *k* = *number of variables* and *v* = *number of values per variable*. Table 6 shows the combination coverage of an equivalent number of randomly generated tests for t=2,3,4. For example, row 2 shows that a covering array with 30 tests covers all 2-way combinations for 10 variables with 4 values each, but 30 randomly generated tests cover only 84.6% of all 2-way combinations.

The coverage provided by a covering array versus a random test suite of the same size varies considerably with different configurations. An important practical consideration in comparing combinatorial with random testing is the effectiveness of the covering array generator. Algorithms have a wide range in the size of covering arrays they produce, but all are designed to produce the smallest array possible that covers all *t*-way combinations. It is not uncommon for the better algorithms to produce arrays that are more than 50% smaller than other algorithms. Comparisons show that there is no uniformly “best” covering array algorithm [10]. Algorithms vary greatly in the size of combinatorial test suites they produce, so the comparable random test suites will also vary in the number of tests. Random testing may produce results similar to combinatorial tests produced by an algorithm that generates a larger, sub-optimal covering array, because the correspondingly larger random test set has a greater probability of covering the *t*-way combinations.

A covering array algorithm that produces a compact array, i.e., a minimal number of tests, for *t*-way combinations may also include fewer (*t*+1)-way combinations because there are fewer tests. Note that at t=2 (pairwise), an equal sized random test set covers more 4-way and 5-way combinations, which may explain why the random tests detected more deadlocks than the t=2 covering array. Almost paradoxically, a sub-optimal algorithm that produces a larger covering array may be more effective because the larger array is statistically more likely to include *t*+1, *t*+2, and higher degree interaction tests as a byproduct of the test generation. This result demonstrates that the smallest possible array is not necessarily best for testing purposes if higher strength interactions are not also tested. It also suggests that covering array generation algorithms that fill "don't care" values (those for which all combinations have already been covered) with random values may provide better test results by covering a larger number of *t*+1, *t*+2, and higher degree combinations.

Table 4: Combination coverage of IPOG t-way tests

t	2-way	3-way	4-way	5-way	Avg
2	1.00	.758	.429	.217	0.601
3	1.00	1.00	.924	.709	0.908
4	1.00	1.00	1.00	.974	0.994

Table 5: Combination coverage, random tests

size = t-way	2-way	3-way	4-way	5-way	Avg
2	.940	.735	.499	.306	0.620
3	1.00	.942	.917	.767	0.906
4	1.00	1.00	.965	.974	0.985

Table 6: Combination coverage of an equivalent number of random tests

Vars	Vals / Var	IPOG tests t=2	Rand 2-way covg	IPOG tests t=3	Rand 3-way covg	IPOG tests t=4	Rand 4-way covg
10	2	10	94.1	20	94.3	42	93.2
10	4	30	84.6	151	90.6	657	92.3
10	6	66	85.6	532	91.6	3843	94.8
10	8	117	83.8	1214	90.6	12010	94.7
10	10	172	82.1	2367	90.6	29231	94.6
15	2	10	93.9	24	96.2	58	97.5
15	4	33	88.1	179	94.1	940	97.5
15	6	77	88.6	663	95.4	5243	98.2
15	8	125	86.1	1551	95.2	16554	98.2
15	10	199	86.4	3000	95.0	40233	98.2
20	2	12	96.5	27	97.3	66	98.6
20	4	37	90.9	209	96.2	1126	98.8
20	6	86	91.3	757	97.0	6291	99.2
20	8	142	91.3	1785	96.9	19882	99.2
20	10	215	88.4	3463	96.9	48374	99.2
25	2	12	95.9	30	98.5	74	99.2
25	4	39	92.1	233	97.5	1320	99.4
25	6	89	91.8	839	97.9	7126	99.6
25	8	148	90.3	1971	97.9	22529	99.6
25	10	229	90.0	3823	97.8	54856	99.6

Now consider the size of a random test set required to provide 100% combination coverage. Table 7 gives the ratio of randomly generated tests to combinatorial tests for the variable/value combinations. For example, for 10 variables with 2 values each, random generation requires 1.80, 3.05, and 3.57 times as many tests as a covering array to cover all combinations at t=2, 3, and 4 respectively. For most covering array algorithms, the difficulty of finding tests with high coverage increases as tests are generated. Thus even if a randomly generated test set provides better than 99% of the coverage of an equal sized covering array, it should not be concluded that only a few more tests are needed for the random set to provide 100% coverage. Table 7 shows that the ratio of random to combinatorial test set size for 100% coverage exceeds 3 in most cases, with average ratios of 3.9, 3.8, and 3.2 at *t* = 2, 3, and 4 respectively. In other words, using random tests to obtain coverage of all t-way combinations required more than three times as many tests as were needed when using a covering array. Thus combinatorial testing offers a significant efficiency advantage over random testing if the goal is 100% combination coverage.

Table 7: Ratio of random to combinatorial tests for 100% combination coverage

Var	Vals/ var	2-way Tests		3-way Tests		4-way Tests	
		IPOG Tests	Ratio	IPOG Tests	Ratio	IPOG Tests	Ratio
10	2	10	1.80	20	3.05	42	3.57
10	4	30	4.83	151	6.05	657	3.43
10	6	66	5.80	532	3.73	3843	3.48
10	8	117	4.26	1214	4.46	12010	4.39
10	10	172	4.70	2367	4.94	29231	4.71
15	2	10	2.00	24	2.17	58	2.24
15	4	33	3.67	179	3.75	940	2.73
15	6	77	3.82	663	3.79	5243	3.26
15	8	125	4.41	1551	4.36	16554	3.66
15	10	199	4.72	3000	5.08	40233	3.97
20	2	12	1.92	27	2.59	66	2.12
20	4	37	3.78	209	2.98	1126	3.35
20	6	86	3.35	757	3.39	6291	2.99
20	8	142	4.44	1785	4.73	19882	3.00
20	10	215	4.78	3463	4.04	48374	3.25
25	2	12	2.83	30	2.33	74	2.35
25	4	39	3.08	233	3.39	1320	2.67
25	6	89	3.67	839	3.44	7126	2.75
25	8	148	5.71	1971	3.76	22529	2.72
25	10	229	4.50	3823	4.32	54856	3.50
Ratio Avg.			3.90		3.82		3.21

The analysis suggests two significant advantages for combinatorial methods in simulations where interactions between input variables are likely to be important:

Significantly fewer tests required to provide 100% combination coverage for a particular interaction strength. Depending on problem size, random generation requires approximately 2 to 6 times as many test inputs as a covering array to cover all combinations (Table 7). While random generation will cover a significant portion of the data space, sometimes 99% or more (Table 6), this may often not be adequate in practice. The network simulation described in previous sections illustrates that combinatorial methods can detect rare interactions that may be missed with an equal number of random inputs.

Better coverage of higher strength interactions. As shown in Table 4, a covering array for interaction strength *t* is likely to provide better coverage of *t*+1, *t*+2, etc. combinations than an equal number of random tests. This characteristic provides a greater chance of detecting events triggered by rare combinations.

5. Conclusions

For the simulation program tested in this study, pairwise tests detected slightly fewer deadlocks than an equal number of random tests, but 4-way combinatorial testing produced better results than an equal number of random tests. Analyzing the random test sets suggests a number of reasons for these results. Although pairwise tests covered all 2-way combinations and an equal number of random tests covered fewer, the random tests covered more 4-way and 5-way combinations, and thus had a greater probability of triggering deadlocks that depended on 4-way or 5-way interactions. However, the 4-way combinatorial tests covered significantly more 4-way combinations (100% vs. 96%) and also provided equal 5-way coverage compared with the corresponding random test set, and found more deadlocks as well.

This result demonstrated that the smallest possible array is not necessarily best for testing purposes if higher strength interactions are not also tested. When using *t*-way combinatorial testing, it can be helpful to evaluate the test set for coverage of *t*+1 and higher interaction strengths. Methods of combining combinatorial and random tests may also be effective, as proposed in [2],[1]. These results also suggest that covering array algorithms may provide better test results by filling "don't care" values with random (rather than constant, sequential, or other non-random) values.

Note: Reference to commercial products or trademarks does not imply endorsement by NIST, nor that such products are necessarily best suited to any purpose.

References

1. Bell, K.Z. and M.A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. Proc. ITI Third IEEE Int Conf on Information & Communications Technology, pages 221–235, Cairo, Egypt, Dec. 2005.
2. Bell, K.Z., Optimizing Effectiveness and Efficiency of Software Testing: a Hybrid Approach, PhD Dissertation, North Carolina State University, 2006.
3. Burr K., and W. Young, Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage, Int Conf on Software Testing, Analysis, and Review (STAR), San Diego, CA, October, 1998
4. Burroughs, K., A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In Proceedings of the IEEE Intl Conf on Comm. (Supercomm/ICC'94), May 1-5, New Orleans, Louisiana, USA. IEEE, May 1994, pp. 745-752
5. Cohen, D. M., S. R. Dalal, J. Parelius, G. C. Patton The Combinatorial Design Approach to Automatic Test Generation IEEE Software, 13, n. 5, pp. 83-87, Sept 1996
6. Dunietz, S., W. K. Ehrlich, B. D. Szablak, C. L. Mallows, A. Iannino. Applying design of experiments to software testing Proceedings of the Intl. Conf. on Software Engineering, (ICSE '97), 1997, pp. 205-215, New York
7. Kuhn, D. R., D. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," IEEE Transactions Software Engineering, 30(6):418-421, 2004.
8. Kuhn, D. R. and V. Okun, "Pseudo-exhaustive Testing for Software," Proceedings of 30th NASA/IEEE Software Engineering Workshop, pp. 153-158, 2006.
9. Kuhn, D.R., M.J. Reilly, An Investigation of the Applicability of Design of Experiments to Software Testing, 27th NASA/IEEE Software Eng. Workshop, NASA Goddard Space Flight Center, 4-6 Dec. ,2002 .
10. Lei, Y., R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing", Software Testing, Verification, and Reliability. (Nov 29 2007, DOI: 10.1002/stvr.381)
11. Lei, Y., R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, "IPOG - a General Strategy for t-way Testing", IEEE Engineering of Computer Based Systems Conf, 2007.
12. Kobayashi, N., T. Tsuchiya, T. Kikuno, "Applicability of Non-Specification Based Approaches to Logic Testing for Software", *Proc. 2001 International Conference on Dependable Systems and Networks*, IEEE, pp. 337 – 346.
13. Pardo, F., JSimured - Simulador de Redes de Multicomputadores Paralelo, University of Valencia, May, 2005. http://simured.uv.es/doc/memoria.pdf
14. Pretschner, A., Tejeddine Mouelhi, Yves Le Traon. Model Based Tests for Access Control Policies, *2008 International Conference on Software Testing, Verification, and Validation* pp. 338-347
15. Wallace, D. R. and D. R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," International Journal of Reliability, Quality and Safety Engineering, 8(4):351-371, 2001.
16. Williams, A.W., R.L. Probert. A practical strategy for testing pair-wise coverage of network interfaces The Seventh International Symposium on Software Reliability Engineering (ISSRE '96) p. 246

Random vs. Combinatorial Methods for Discrete Event Simulation of a Grid Computer Network

D. Richard Kuhn*, Raghu Kacker*, Yu Lei**

**National Institute of Standards and Technology, ** University of Texas, Arlington*

kuhn@nist.gov, raghu.kacker@nist.gov, ylei@uta.edu

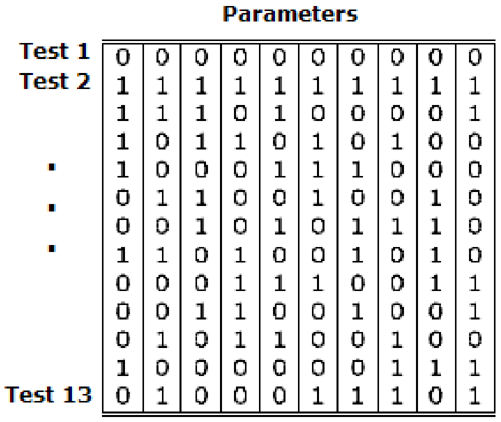
Abstract: This study compared random and *t*-way combinatorial inputs of a network simulator, to determine if these two approaches produce significantly different deadlock detection for varying network configurations. Modeling deadlock detection is important for analyzing configuration changes that could inadvertently degrade network operations, or to determine modifications that could be made by attackers to deliberately induce deadlock. Discrete event simulation of a network may be conducted using random generation of inputs. In this study, we compare random with combinatorial generation of inputs. Combinatorial (or *t*-way) testing requires every combination of any *t* parameter values to be covered by at least one test. Combinatorial methods can be highly effective because empirical data suggest that nearly all failures involve the interaction of a small number of parameters (1 to 6). Thus, for example, if all deadlocks involve at most 5-way interactions between *n* parameters, then exhaustive testing of all *n*-way interactions adds no additional information that would not be obtained by testing all 5-way interactions. While the maximum degree of interaction between parameters involved in the deadlocks clearly cannot be known in advance, covering all *t*-way interactions may be more efficient than using random generation of inputs. In this study we tested this hypothesis for *t* = 2, 3, and 4 for deadlock detection in a network simulation. Achieving the same degree of coverage provided by 4-way tests would have required approximately 3.2 times as many random tests; thus combinatorial methods were more efficient for detecting deadlocks involving a higher degree of interactions. The paper reviews explanations for these results and implications for modeling and simulation.

1 Background

A number of studies have shown combinatorial methods to be highly effective for software testing (e.g., [3],[6],[16],[8]. The effectiveness of combinatorial test methods rests on the observation that a significant number of events in software are triggered only by the interaction of two or more variable values. By including tests for all 2-way, 3-way, etc., interactions, the test set should be able to detect events that occur only with complex interactions. The complexity of discrete event simulation suggests that, as with software testing, combinatorial methods may be effective for finding events triggered only by rare multi-way interactions of input values. In this paper, we compare the effectiveness of combinatorial versus random generation of inputs in a grid computer network simulation for finding configurations that lead to deadlock.

The key enabler in combinatorial testing is a *covering array* that covers all *t*-way combinations of parameter values, for a desired strength *t*. Covering arrays are combinatorial objects that represent interaction test suites. A covering array, $CA(N;t,k,v)$, is an *N* x *k* array, where *k* is the number of variables, and *v* is the number of possible values for each variable such that in every *N* x *t* subarray, each *t*-tuple occurs at least once, then *t* is the *strength* of the coverage of interactions. Each row of a covering array represents a test, with one column for each parameter that is varied in testing. Collectively, the rows of the array include every *t*-way combination of parameter values at least once. For example, Figure 1 shows a covering array that includes all 3-way combinations of binary values for 10 parameters. Each row corresponds to one test, and each column gives the values for a particular parameter. It can be seen that any three columns in any order contain all eight possible combinations (000, 001, 010, 011, 100, 101, 110, 111) of the parameter values. Collectively, this set of tests will exercise all 3-way combinations of input values in only 13 tests, as compared with 1,024 for exhaustive coverage.

The primary goal in the simulation is to study the behavior of the system with different input configurations. For example, a network simulation may investigate the effect of configurations on packet rate, delay, or potential for deadlock in the network, just as.a production line simulation may study the effects of changing line speed, interconnection between workstations, and buffer size on the number of items that can be produced per hour.



	Parameters									
Test 1	0	0	0	0	0	0	0	0	0	0
Test 2	1	1	1	1	1	1	1	1	1	1
	1	1	1	0	1	0	0	0	0	1
	1	0	1	1	0	1	0	1	0	0
.	1	0	0	0	1	1	1	0	0	0
.	0	1	1	0	0	1	0	0	1	0
	0	0	1	0	1	0	1	1	1	0
.	1	1	0	1	0	0	1	0	1	0
	0	0	0	1	1	1	0	0	1	1
	0	0	1	1	0	0	1	0	0	1
	0	1	0	1	1	0	0	1	0	0
	1	0	0	0	0	0	0	1	1	1
Test 13	0	1	0	0	0	1	1	1	0	1

Figure 1: 3-way covering array for 10 parameters with 2 values each

In this study we compare random and combinatorial testing of a network simulator, to determine if these two test approaches produce significantly different deadlock detection in the simulation. Using deadlocks as events of interest makes evaluating program responses straightforward and unambiguous. Numerical results such as packet rates or delays are not considered, but could be the subject of a future investigation. The two test modes – random or combinatorial – are compared using a standard two-tailed t-test for statistical significance.

2 Experimental Evaluation

This work investigates the hypothesis that combinatorial test suites will detect significantly more deadlocks than random test suites of the same size, for interaction strengths of t = 2, 3, 4.

Independent and Dependent Variables: The independent variable in this study is the type of testing used, either *t*-way combinatorial or random. The dependent variable is the number of deadlocks detected.

Subject Application and Test Suites: Software under test for the experiment was Simured [13], a multicomputer network simulator developed at the University of Valencia. The software is available in C++ and Java versions, for both Linux and Windows. The core command line code (not including user interface or graphical display) consists of 2,131 lines of C++. Simured provides a simulation of the switching and routing layers for a multicomputer, allowing the user to study grid computer configurations to investigate the effect of topologies and configurable parameters on routing, timing, and other variables of interest. We used the C++ command line version of this software, compiled with gcc and run on 64-bit processors under Red Hat Enterprise Linux V4. No modifications were made to the Simured software.

Table 1: Simured configuration parameters and test values used

Parameter		Values
1	DIMENSIONS	1,2,4,6,8
2	NODOSDIM	2,4,6
3	NUMVIRT	1,2,3,8
4	NUMVIRTINJ	1,2,3,8
5	NUMVIRTEJE	1,2,3,8
6	LONBUFFER	1,2,4,6
7	NUMDIR	1,2
8	FORWARDING	0,1
9	PHYSICAL	t, f
10	ROUTING	0,1,2,3
11	DELFIFO	1,2,4,6
12	DELCROSS	1,2,4,6
13	DELCHANNEL	1,2,4,6
14	DELSWITCH	1,2,4,6

Simured provides a set of 14 parameters that can be set to a variety of values in a configuration file that is read by the simulator. Parameters and possible values used are shown in Table 1. The total number of possible configurations with these parameter values is 3.1 x 10^7. Larger values are possible for a number of parameters, but would require extensive run time on a large system.

Evaluation Metrics: Test suites were evaluated according to the number of deadlocks detected. We also compare the percentage of *t*-way combinations covered for the random test suites of equal size, and determine the number of random tests needed to provide 100% coverage of the respective *t*-way combinations. (By definition, a covering array provides 100% coverage of *t*-way combinations.)

Threats to Validity: Clearly there are limitation on the extent to which these results can be generalized to other applications. While previous comparisons of combinatorial and random testing focused on fault detection, this study evaluates these methods with respect to deadlock detection in a simulation. Some implications of this difference are discussed in the analysis of results, in Section 4.2. A second difference is the nature of the software under test. Simured is a small but complex program that is not assumed to have characteristics similar to other application domains. Network simulation requires extensive calculations for statistics such as packet transmission rates and delays, and is not directly comparable to other types of software.

While the issues raised above should be considered in evaluating results, we believe that the experiment has identified a number of factors that can be usefully considered when deciding whether to use random or combinatorial testing for a particular problem.

3 Testing Procedure

Covering arrays that include all *t*-way combinations for *t* = 2, 3, and 4 were generated using the IPOG algorithm [11], which produces compact test suites. Test suites for the configuration shown in 0 included 28, 161, and 752 tests for *t* = 2, 3, and 4 respectively. Random test suites matching the sizes of the 2, 3, and 4-way combinatorial test suites were produced using the standard C library *rand()* function, producing one test at a time with a call to *rand()* for each variable value. In generating random test sets, the *rand()* function was initialized with a call to *srand()* to seed the pseudo-random number generator from the system clock. From these tests, configuration files were generated for Simured and the command line version of Simured invoked with each configuration file.

Each test set was executed for 500, 1000, 2000, 4000, and 8000-packet simulation runs. For combinatorial testing, one test suite run was conducted for each of the five packet counts and three interaction levels (28, 161, and 752 tests, for a total of 4,705 simulations). Random generation produces a different test set with each test generation run. For random testing, eight runs at each combination of packet count and interaction level were conducted (37,640 simulations), and the average deadlock detection calculated.

4 Results and Analysis

4.1 Test Results

Results for the two test modes were compared with a standard t-test for paired samples. Table 2 shows the number of deadlocks detected using tests produced from IPOG versus the average number of deadlocks detected with an equal number of randomly generated tests. Values for random test detection represent the average of eight runs with randomly generated tests at each combination of interaction level and packet count. Table 3 gives the two-tailed probability of a difference between the numbers of deadlocks detected by combinatorial and random testing.

Table 2: Deadlocks, combinatorial vs. random

Deadlocks Detected – combinatorial						
t	Tests	Packets				
		500	1000	2000	4000	8000
2	28	0	0	0	0	0
3	161	2	3	2	3	3
4	752	14	14	14	14	14
Average Deadlocks Detected – random						
t	Tests	Packets				
		500	1000	2000	4000	8000
2	28	0.63	0.25	0.75	0. 50	0. 75
3	161	3.00	3.00	3.00	3.00	3.00
4	752	10.13	11.75	10.38	13.00	13.25

Table 3: t-test results for difference between random and IPOG generated tests

Interaction strength	**Two-tailed probability**
2	.0035
3	.1778
4	.0235

For pairwise testing (*t* = 2), combinatorial testing detected slightly fewer deadlocks than an equal number of random tests, and the difference is statistically significant. At interaction strength *t* = 3 the difference between the two test methods is not statistically significant. At *t* = 4, however, the covering arrays produced by IPOG detected significantly more deadlocks than an equal number of random tests. In the next section we consider some possible reasons for the variation in effectiveness of these two test methods. Two important considerations should be noted about the difference in deadlocks detected:

combinatorial methods found more deadlock configurations, but also consistently found 14 deadlocks for the most complex (4-way) interactions, while there was a great degree of variation among the random configurations.

4.2 Analysis of Results

In considering explanations for the results, we first note that there can be a number of differences between the simulations conducted in this work and software testing in other application domains. In many applications, such as databases or web applications, different parameter values may result in different execution paths within an application, but the amount and complexity of processing is often similar for many different inputs. Network simulation, by contrast, may exhibit wide variations in processing depending on whether the input configuration is a small network of simple topology, or a large, complex one. This difference was observed in widely varying run times (not reported in this paper), and may also contribute to the distribution of deadlocks detected at the three interaction levels. Previous work (see Section 1) has found that increasing values of *t* detect progressively fewer faults, even in cases where combinatorial testing performed no better than random tests. Pairwise testing (*t*=2) often detected 70% to more than 90% of faults, while 3-way tests found roughly 10% to 20% of faults, and 4-way to 6-way tests typically detected less than 5%. This distribution is essentially reversed for the Simured testing (see Table 2), with 0%, 18%, and 82% of deadlocks detected at *t*=2, 3, and 4 respectively. This result is not unexpected. Faults can be triggered by combinations of any of the variables in a program. Even though a large set of variables may be directly or indirectly involved in triggering deadlocks, the set can be expected to be much smaller than the total number of variables in a program. With deadlocks occurring in roughly 2% of simulation runs, larger test sets would be expected to locate more deadlocks.

In addition to the "reverse" relationship between deadlock detection and interaction strength, another interesting finding was that pairwise tests detected slightly fewer deadlocks than the same number of random tests. Careful analysis shows that there is in fact a combinatorial explanation for this result, which we discuss in the remainder of this section.

Because a significant percentage of events can only be triggered by the interaction of two or more variables, one consideration in comparing random and combinatorial testing is the degree to which random testing covers particular *t*-way combinations. Any test set will also cover a certain proportion of possible (*t*+1)-way, (*t*+2)-way, etc. combinations as well. Tables 4 and 5 compare this coverage for the Simured test inputs.

We also analyzed the average percentage of *t*-way combinations covered by 100 randomly generated test sets of the same size as a *t*-way covering array generated by IPOG, for various combinations of *k = number of variables* and *v = number of values per variable*. Table 6 shows the combination coverage of an equivalent number of randomly generated tests for t=2,3,4. For example, row 2 shows that a covering array with 30 tests covers all 2-way combinations for 10 variables with 4 values each, but 30 randomly generated tests cover only 84.6% of all 2-way combinations.

The coverage provided by a covering array versus a random test suite of the same size varies considerably with different configurations. An important practical consideration in comparing combinatorial with random testing is the effectiveness of the covering array generator. Algorithms have a wide range in the size of covering arrays they produce, but all are designed to produce the smallest array possible that covers all *t*-way combinations. It is not uncommon for the better algorithms to produce arrays that are more than 50% smaller than other algorithms. Comparisons show that there is no uniformly "best" covering array algorithm [10]. Algorithms vary greatly in the size of combinatorial test suites they produce, so the comparable random test suites will also vary in the number of tests. Random testing may produce results similar to combinatorial tests produced by an algorithm that generates a larger, sub-optimal covering array, because the correspondingly larger random test set has a greater probability of covering the *t*-way combinations.

A covering array algorithm that produces a compact array, i.e., a minimal number of tests, for *t*-way combinations may also include fewer (*t*+1)-way combinations because there are fewer tests. Note that at t=2 (pairwise), an equal sized random test set covers more 4-way and 5-way combinations, which may explain why the random tests detected more deadlocks than the t=2 covering array. Almost paradoxically, a sub-optimal algorithm that produces a larger covering array may be more effective because the larger array is statistically more likely to include *t*+1, *t*+2, and higher degree interaction tests as a byproduct of the test generation. This result demonstrates that the smallest possible array is not necessarily best for testing purposes if higher strength interactions are not also tested. It also suggests that covering array generation algorithms that fill "don't care" values (those for which all combinations have already been covered) with random values may provide better test results by covering a larger number of *t*+1, *t*+2, and higher degree combinations.

Table 4: Combination coverage of IPOG t-way tests

t	2-way	3-way	4-way	5-way	Avg
2	1.00	.758	.429	.217	0.601
3	1.00	1.00	.924	.709	0.908
4	1.00	1.00	1.00	.974	0.994

Table 5: Combination coverage, random tests

size = t-way	2-way	3-way	4-way	5-way	Avg
2	.940	.735	.499	.306	0.620
3	1.00	.942	.917	.767	0.906
4	1.00	1.00	.965	.974	0.985

Table 6: Combination coverage of an equivalent number of random tests

Vars	Vals / Var	IPOG tests t=2	Rand 2-way covg	IPOG tests t=3	Rand 3-way covg	IPOG tests t=4	Rand 4-way covg
10	2	10	94.1	20	94.3	42	93.2
10	4	30	84.6	151	90.6	657	92.3
10	6	66	85.6	532	91.6	3843	94.8
10	8	117	83.8	1214	90.6	12010	94.7
10	10	172	82.1	2367	90.6	29231	94.6
15	2	10	93.9	24	96.2	58	97.5
15	4	33	88.1	179	94.1	940	97.5
15	6	77	88.6	663	95.4	5243	98.2
15	8	125	86.1	1551	95.2	16554	98.2
15	10	199	86.4	3000	95.0	40233	98.2
20	2	12	96.5	27	97.3	66	98.6
20	4	37	90.9	209	96.2	1126	98.8
20	6	86	91.3	757	97.0	6291	99.2
20	8	142	91.3	1785	96.9	19882	99.2
20	10	215	88.4	3463	96.9	48374	99.2
25	2	12	95.9	30	98.5	74	99.2
25	4	39	92.1	233	97.5	1320	99.4
25	6	89	91.8	839	97.9	7126	99.6
25	8	148	90.3	1971	97.9	22529	99.6
25	10	229	90.0	3823	97.8	54856	99.6

Now consider the size of a random test set required to provide 100% combination coverage. Table 7 gives the ratio of randomly generated tests to combinatorial tests for the variable/value combinations. For example, for 10 variables with 2 values each, random generation requires 1.80, 3.05, and 3.57 times as many tests as a covering array to cover all combinations at t=2, 3, and 4 respectively. For most covering array algorithms, the difficulty of finding tests with high coverage increases as tests are generated. Thus even if a randomly generated test set provides better than 99% of the coverage of an equal sized covering array, it should not be concluded that only a few more tests are needed for the random set to provide 100% coverage. Table 7 shows that the ratio of random to combinatorial test set size for 100% coverage exceeds 3 in most cases, with average ratios of 3.9, 3.8, and 3.2 at *t* = 2, 3, and 4 respectively. In other words, using random tests to obtain coverage of all t-way combinations required more than three times as many tests as were needed when using a covering array. Thus combinatorial testing offers a significant efficiency advantage over random testing if the goal is 100% combination coverage.

Table 7: Ratio of random to combinatorial tests for 100% combination coverage

Var	Vals/ var	2-way Tests		3-way Tests		4-way Tests	
		IPOG Tests	Ratio	IPOG Tests	Ratio	IPOG Tests	Ratio
10	2	10	1.80	20	3.05	42	3.57
10	4	30	4.83	151	6.05	657	3.43
10	6	66	5.80	532	3.73	3843	3.48
10	8	117	4.26	1214	4.46	12010	4.39
10	10	172	4.70	2367	4.94	29231	4.71
15	2	10	2.00	24	2.17	58	2.24
15	4	33	3.67	179	3.75	940	2.73
15	6	77	3.82	663	3.79	5243	3.26
15	8	125	4.41	1551	4.36	16554	3.66
15	10	199	4.72	3000	5.08	40233	3.97
20	2	12	1.92	27	2.59	66	2.12
20	4	37	3.78	209	2.98	1126	3.35
20	6	86	3.35	757	3.39	6291	2.99
20	8	142	4.44	1785	4.73	19882	3.00
20	10	215	4.78	3463	4.04	48374	3.25
25	2	12	2.83	30	2.33	74	2.35
25	4	39	3.08	233	3.39	1320	2.67
25	6	89	3.67	839	3.44	7126	2.75
25	8	148	5.71	1971	3.76	22529	2.72
25	10	229	4.50	3823	4.32	54856	3.50
Ratio Avg.			3.90		3.82		3.21

The analysis suggests two significant advantages for combinatorial methods in simulations where interactions between input variables are likely to be important:

Significantly fewer tests required to provide 100% combination coverage for a particular interaction strength. Depending on problem size, random generation requires approximately 2 to 6 times as many test inputs as a covering array to cover all combinations (Table 7). While random generation will cover a significant portion of the data space, sometimes 99% or more (Table 6), this may often not be adequate in practice. The network simulation described in previous sections illustrates that combinatorial methods can detect rare interactions that may be missed with an equal number of random inputs.

Better coverage of higher strength interactions. As shown in Table 4, a covering array for interaction strength *t* is likely to provide better coverage of *t*+1, *t*+2, etc. combinations than an equal number of random tests. This characteristic provides a greater chance of detecting events triggered by rare combinations.

5. Conclusions

For the simulation program tested in this study, pairwise tests detected slightly fewer deadlocks than an equal number of random tests, but 4-way combinatorial testing produced better results than an equal number of random tests. Analyzing the random test sets suggests a number of reasons for these results. Although pairwise tests covered all 2-way combinations and an equal number of random tests covered fewer, the random tests covered more 4-way and 5-way combinations, and thus had a greater probability of triggering deadlocks that depended on 4-way or 5-way interactions. However, the 4-way combinatorial tests covered significantly more 4-way combinations (100% vs. 96%) and also provided equal 5-way coverage compared with the corresponding random test set, and found more deadlocks as well.

This result demonstrated that the smallest possible array is not necessarily best for testing purposes if higher strength interactions are not also tested. When using *t*-way combinatorial testing, it can be helpful to evaluate the test set for coverage of *t*+1 and higher interaction strengths. Methods of combining combinatorial and random tests may also be effective, as proposed in [2],[1]. These results also suggest that covering array algorithms may provide better test results by filling "don't care" values with random (rather than constant, sequential, or other non-random) values.

Note: Reference to commercial products or trademarks does not imply endorsement by NIST, nor that such products are necessarily best suited to any purpose.

References

1. Bell, K.Z. and M.A. Vouk. On effectiveness of pairwise methodology for testing network-centric software. Proc. ITI Third IEEE Int Conf on Information & Communications Technology, pages 221–235, Cairo, Egypt, Dec. 2005.
2. Bell, K.Z., Optimizing Effectiveness and Efficiency of Software Testing: a Hybrid Approach, PhD Dissertation, North Carolina State University, 2006.
3. Burr K., and W. Young, Combinatorial Test Techniques: Table-Based Automation, Test Generation, and Test Coverage, Int Conf on Software Testing, Analysis, and Review (STAR), San Diego, CA, October, 1998
4. Burroughs, K., A. Jain, and R. L. Erickson. Improved quality of protocol testing through techniques of experimental design. In Proceedings of the IEEE Intl Conf on Comm. (Supercomm/ICC'94), May 1-5, New Orleans, Louisiana, USA. IEEE, May 1994, pp. 745-752
5. Cohen, D. M., S. R. Dalal, J. Parelius, G. C. Patton The Combinatorial Design Approach to Automatic Test Generation IEEE Software, 13, n. 5, pp. 83-87, Sept 1996
6. Dunietz, S., W. K. Ehrlich, B. D. Szablak, C. L. Mallows, A. Iannino. Applying design of experiments to software testing Proceedings of the Intl. Conf. on Software Engineering, (ICSE '97), 1997, pp. 205-215, New York
7. Kuhn, D. R., D. Wallace, and A. Gallo, "Software Fault Interactions and Implications for Software Testing," IEEE Transactions Software Engineering, 30(6):418-421, 2004.
8. Kuhn, D. R. and V. Okun, "Pseudo-exhaustive Testing for Software," Proceedings of 30th NASA/IEEE Software Engineering Workshop, pp. 153-158, 2006.
9. Kuhn, D.R., M.J. Reilly, An Investigation of the Applicability of Design of Experiments to Software Testing, 27th NASA/IEEE Software Eng. Workshop, NASA Goddard Space Flight Center, 4-6 Dec. ,2002 .
10. Lei, Y., R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, "IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing", Software Testing, Verification, and Reliability. (Nov 29 2007, DOI: 10.1002/stvr.381)
11. Lei, Y., R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, "IPOG - a General Strategy for t-way Testing", IEEE Engineering of Computer Based Systems Conf, 2007.
12. Kobayashi, N., T. Tsuchiya, T. Kikuno, "Applicability of Non-Specification Based Approaches to Logic Testing for Software", *Proc. 2001 International Conference on Dependable Systems and Networks*, IEEE, pp. 337 – 346.
13. Pardo, F., JSimured - Simulador de Redes de Multicomputadores Paralelo, University of Valencia, May, 2005. http://simured.uv.es/doc/memoria.pdf
14. Pretschner, A., Tejeddine Mouelhi, Yves Le Traon. Model Based Tests for Access Control Policies, *2008 International Conference on Software Testing, Verification, and Validation* pp. 338-347
15. Wallace, D. R. and D. R. Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 Years of Recall Data," International Journal of Reliability, Quality and Safety Engineering, 8(4):351-371, 2001.
16. Williams, A.W., R.L. Probert. A practical strategy for testing pair-wise coverage of network interfaces The Seventh International Symposium on Software Reliability Engineering (ISSRE '96) p. 246